3. The Necessary Theory Part 3/3

Table of Contents

- Introduction
- Heaps [1]
- Handles [2]
- Exceptions [3]
- Basic Windows Ring3 Internal Structures
- Windows APIs
- Types of Reversing Tools
- Conclusions
- References

Introduction

The third chapter of this course aims to offer some extra theoretical knowledge necessary for the rest of the course.

During this chapter we will briefly discuss the concept of heaps, handles, exceptions, and some basic Windows Ring3 Internal structures. We'll also cover a broad overview of Windows APIs.

Finally, we'll go through the most common reversing tools used today for software reverse engineering.

Some of the concepts that we will be discussing during this chapter are tightly integrated with another very important part of the reverse engineering process: the various anti-reversing tricks employed. We'll cover these later on.

Heaps [1]

When we talk about heaps, we are referring to memory areas dynamically allocated at runtime, used to store data that doesn't have a fixed size or data that can't fit inside the stack.

In order to get a grasp on this concept, let's look at a simple example. Let's assume that someone is creating an application for file manipulation and because of its organization, the entire file need to be loaded into memory somewhere.

Of course, the developer can't predict the size of every file that could be loaded, but at the same time, initialising a huge static array able to theoretically fit any file is not an acceptable option, either. His application would be using much more physical memory than necessary, which would affect the performance of the system on which the application is running.

Because of this, we need dynamic memory allocation in order to be able to fit the selected file into memory, regardless of size, while at the same time, not wasting any more physical memory than necessary.

Of course, such types of applications could perform other checks, such as checking the amount of physical memory available, etc. This is out of the scope of this course, but this example gives you an idea of what heaps are and why we need them.

Handles [2]

Handles can be considered references to various resources, and they are used by the operating system in order to control the resource access (read, write, etc).

A simple example regarding handles in Windows has to do with file manipulation. In order to access a file, you first need to create a handle to that file using a Windows API, such as the CreateFile API, which returns a handle - in this case, a number that identifies the specific resource with the requested access rights to it.

Exceptions [3]

An exception is a specific type of event that occurs during the execution of an application.

Exceptions are normally associated with specific exception handlers, which are code blocks dedicated to handle that type of event correctly. Since exceptions can also be caused by programming errors, it is important to have an exception handler that will prevent the application from crashing unexpectedly.

There are two types of exceptions: hardware and software.

Hardware exceptions are normally caused by the execution of a bad sequence of instructions such as a division by zero or an attempt to access either an invalid memory location or a memory location that the application doesn't have the necessary access rights to. Hardware exceptions are initiated by the CPU. The Windows OS is able to recognize hardware exceptions and map them to a list of predefined constants or exceptions codes.

On the other hand, software exceptions are generated by the application (or the operating system itself) and can be used in order to flag certain conditions which user defined exception codes.

Windows implements its own mechanism to handle exceptions which is called Structured Exception Handling (SEH). SEH gives the application the ability to handle both software and hardware exceptions in the same manner, and allows the application to have full control over any exceptions raised during its execution.

Take a look at the following example. Here, the application is setting its own custom exception handler and then forcing a division by zero exception.

```
00401025   68 60104000        PUSH   exceptio.00401060 ← push address of handler.
0040102A   64:FF35 0000000   PUSH   DWORD PTR FS:[0] ← save address of SEH
chain and...
00401031   64:8925 0000000   MOV    DWORD PTR FS:[0], ESP ← ...substitute it with
our own.
00401038   33C0              XOR    EAX, EAX ← zero out EAX.
0040103A   F7F0              DIV    EAX ← force a division by zero exception. The
execution is now transferred to the __Handler label

0040103C   64:8F05 0000000   POP    DWORD PTR FS:[0] ← we return here when we
exit the handler routine. Restore old address of SEH chain.
00401043   83C4 04           ADD    ESP, 4 ← re-balance the stack, and continue
with normal execution flow.
00401046   68 21204000       PUSH   exceptio.00402021      ; ASCII "Hello world."
0040104B   E8 3C000000       CALL   exceptio.0040108C
00401050   68 2E204000       PUSH   exceptio.0040202E

00401055   E8 32000000       CALL   exceptio.0040108C
0040105A   C3                RET
0040105B   E8 98000000       CALL   exceptio.004010F8

__Handler: (the execution flow is redirected here once the exception occurs)
00401060   68 31204000       PUSH   exceptio.00402031   ; ASCII "I am the exception
Handler."
00401065   E8 22000000       CALL   exceptio.0040108C
0040106A   68 4D204000       PUSH   exceptio.0040204D
0040106F   E8 18000000       CALL   exceptio.0040108C
00401074   8B4424 0C         MOV    EAX, DWORD PTR SS:[ESP+C] ← get address
of CONTEXT structure (see next section for more info about this structure).
00401078   05 B8000000       ADD    EAX, 0B8 ← go to EIP field inside the structure
0040107D   C700 3C104000     MOV    DWORD PTR DS:[EAX], exceptio.0040103C ←
set new EIP to resume execution once exiting from the handler routine.
00401083   B8 00000000       MOV    EAX, 0
00401088   C3                RET ← exit handler.
```

Code Snippet 3.1 SEH Example

The code above would normally display a "Hello World" message on the screen and then exit.

However, the division by zero exception generated changes the execution flow and redirects the flow to inside the handler and then back to normal execution flow. Tricks like this are commonly used as anti-reversing tricks in order to make static and dynamic analysis of code more difficult for both the user and the tools used (Chapter 12).

Basic Windows Ring3 Internal Structures

Let's have a look at two "userland" (Ring3) structures which keep very important information regarding the running process and its threads under execution.

The first one is the THREAD_ENVIRONMENT_BLOCK (TEB) [4]. This structure stores information such as: the addresses of the top and bottom of the current thread's stack, the thread identifier, the identifier of the process the thread belongs to, the code of the last error that occurred during the thread execution, the address of the Thread Local Storage (TLS), the address of the PROCESS_ENVIRONMENT_BLOCK (PEB) [4], which is, coincidentally, the next structure we'll discuss.

The PEB structure is also very important to understand. Some of the information stored in the PEB is: the image base of the process, the address of the loader data structure (PEB_LDR_DATA - which contains pointers to lists regarding the modules loaded by the process) [4], the NtGlobalFlag value, the major and minor versions of the Windows OS< the number of processors available, the BeingDebugged flag and much more.

Some of the information stored inside the PEB can be used for debugger detection, such as the NtGlobalFlag, and the BeingDebugged flag (Chapter 10).

Another important structure is the CONTEXT structure [4]. The system uses this structure to keep track of all necessary CPU state information for a specific thread under execution during internal operations.

For example, it stores the values of the registers which would be necessary to continue the execution of the thread from the correct virtual address in case an event (like an exception) occurs.

Windows APIs

Windows APIs are Ring3 operating system functions.

Their use is well documented, the parameters they receive and their purpose clear so that they can bused by the programmers in order to take advantage of the various functionalities of the operating system (and, of course, to communicate with the kernel functions in a "safe" way.)

In short, they are used as a communication path from the application to the underlying operating system.

Windows APIs Functional Categories:

1. Administration and Management
2. Diagnostics

3. Graphics and Multimedia
4. Networking
5. System Services
6. Windows User Interface

Types of Reversing Tools

During the reverse engineering process, various tools can be used, including:

Hex editor:

This tool actually reads the contents of any file type and shows it in groups of bytes, using a hexadecimal representation of the data. Even if it is not considered a powerful tool on its own any longer, since more advanced code usually incorporate hex editing capabilities, it can still be used in special cases like static analysis of non-executable file formats.

Decompiler:

The purpose of this tool is to attempt to translate low level code (such as machine code) into a high level language which is much easier to understand.

Decompilers are mostly used for executables that run under the java or .Net virtual machines; this is due to the fact that the source code is not really compiled into binary but to an intermediate state between source and machine code, which makes it possible to easily decompile it back to the source state.

There are also some attempts to do the same for native code compiled executables. However, the results are just an indication of how the source may look and it is not going to be 100% accurate.

Disassembler:

A very useful tool used to translate binary into human readable assembly instructions. It can only offer static analysis capabilities, which are usually incorporated within a debugger.

Debugger:

Incorporates hex editing and disassembling capabilities, as well as the possibility to control the execution flow of the application in real time by insertin breakpoints to stop execution on selected instructions.

Debuggers allow reading and writing to the process address space at will and, of course, modifying the assembly instructions as needed.

There are two different types of debuggers, Ring0 (or "kernel mode"), and Ring3 (or "user mode") debuggers. Kernel mode debuggers are more powerful because they actually operate at the kernel level, thus they enable us to access any address in memory.

Furthermore, they are undetectable by most of the commonly used anti-reversing techniques since they actually operate at the system level. This fact that makes them invisible to an application that tries to detect a debugger with Ring3 APIs.

Ring3 debuggers have the disadvantage of being easily detected by various anti-reversing techniques since they operate in user mode as a normal application. This, in the end, turns out to be an advantage for a reverse engineering enthusiast, because this fact allows him to study more about how the anti-reversing techniques work, and learn how much about the underlying operating system.

In addition, through a Ring3 debugger, we can only access memory addresses that are part of the address space of the process that we are debugging.

On the other hand, Ring3 debuggers are all we need most of the time, and usually they are much more user-friendly that the Ring0 ones.

System monitoring tools:

This type of tool is used when we want to monitor various actions of the process under analysis, such as access to various resources like files and registry keys.

WIndows API monitoring tools:

These tools monitor (hook) Windows APIs used by the process under examination. They can be used in order to get a quick overview of the Windows APIs used by any process.

Throughout these first three chapters, we've covered most of the theory necessary to get started with the technical parts of this course. As the course progresses, we will be discussing these concepts in more detail, and also adding to your knowledge with new concepts. This is in order to offer you as much knowledge and understanding about real, practical approaches for software reverse engineering.

Conclusions

In this chapter we discussed some more interesting and necessary concepts including heaps, exceptions, some Windows-specific Ring3 structures, and what Windows APIs are. In addition, we summarized some common types of tools used in reverse engineering.

In the next chapter, we'll continue with some very important concepts lie: Virtual Addresses, Relative Virtual Addresses, Offsets, and an overview of the structure of a Windows executable file.

References

1. Heaps: Pleasures and Pains. https://msdn.microsoft.com/en-us/library/ms810466.aspx
2. Handles and Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724457(v=vs.85).aspx
3. Structured Exception Handling. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx
4. Introduction to NT Internals - Part I. http://www.alex-ionescu.com/